Mathematical Foundations of Data Sciences

Gabriel Peyré
CNRS & DMA
École Normale Supérieure
gabriel.peyre@ens.fr
https://mathematical-tours.github.io
www.numerical-tours.com

November 18, 2020

Chapter 14

Optimization & Machine Learning: Advanced Topics

14.1 Regularization

When the number n of sample is not large enough with respect to the dimension p of the model, it makes sense to regularize the empirical risk minimization problem.

14.1.1 Penalized Least Squares

For the sake of simplicity, we focus here on regression and consider

$$\min_{x\in\mathbb{R}^p} f_\lambda(x) \overset{\text{def.}}{=} \frac{1}{2}\|Ax-y\|^2 + \lambda R(x) \tag{14.1}$$

where $R(x)$ is the regularizer and $\lambda \geqslant 0$ the regularization parameter. The regularizer enforces some prior knowledge on the weight vector x (such as small amplitude or sparsity, as we detail next) and λ needs to be tuned using cross-validation.

We assume for simplicity that R is positive and coercive, i.e. $R(x) \to +\infty$ as $\|x\| \to +\infty$. The following proposition that in the small λ limit, the regularization select a sub-set of the possible minimizer. This is especially useful when $\ker(A) \neq 0$, i.e. the equation $Ax = y$ has an infinite number of solutions.

Proposition 44. *If $(x_{\lambda_k})_k$ is a sequence of minimizers of f_λ, then this sequence is bounded, and any accumulation $x^\star$ is a solution of the constrained optimization problem*

$$\min_{Ax=y} R(x). \tag{14.2}$$

Proof. Let x_0 be so that $Ax_0 = y$, then by optimality of x_{λ_k}

$$\frac{1}{2}\|Ax_{\lambda_k}-y\|^2 + \lambda_k R(x_{\lambda_k}) \leqslant \lambda_k R(x_0). \tag{14.3}$$

Since all the term are positive, one has $R(x_{\lambda_k}) \leqslant R(x_0)$ so that $(x_{\lambda_k})_k$ is bounded by coercivity of R. Then also $\|Ax_{\lambda_k}-y\| \leqslant \lambda_k R(x_0)$, and passing to the limit, one obtains $Ax^\star = y$. And passing to the limit in $R(x_{\lambda_k}) \leqslant R(x_0)$ one has $R(x^\star) \leqslant R(x_0)$ which shows that $x^\star$ is a solution of (14.2). □

14.1.2 Ridge Regression

Ridge regression is by far the most popular regularizer, and corresponds to using $R(x) = \|x\|^2_{\mathbb{R}^p}$. Since it is strictly convex, the solution of (14.1) is unique

$$x_\lambda \overset{\text{def.}}{=} \operatorname*{argmin}_{x\in\mathbb{R}^p} f_\lambda(x) = \frac{1}{2}\|Ax-y\|^2_{\mathbb{R}^n} + \lambda\|x\|^2_{\mathbb{R}^p}.$$

One has

$$\nabla f_\lambda(x) = A^\top(Ax_\lambda - y) + \lambda x_\lambda = 0$$

so that x_λ depends linearly on y and can be obtained by solving a linear system. The following proposition shows that there are actually two alternate formula.

Proposition 45. *One has*

$$x_\lambda = (A^\top A + \lambda \mathrm{Id}_p)^{-1} A^\top y, \tag{14.4}$$
$$= A^\top (AA^\top + \lambda \mathrm{Id}_n)^{-1} y. \tag{14.5}$$

Proof. Denoting $B \overset{\text{def.}}{=} (A^\top A + \lambda \mathrm{Id}_p)^{-1} A^\top$ and $C \overset{\text{def.}}{=} A^\top (AA^\top + \lambda \mathrm{Id}_n)^{-1}$, one has $(A^\top A + \lambda \mathrm{Id}_p)B = A^\top$ while

$$(A^\top A + \lambda \mathrm{Id}_p)C = (A^\top A + \lambda \mathrm{Id}_p)A^\top (AA^\top + \lambda \mathrm{Id}_n)^{-1} = A^\top (AA^\top + \lambda \mathrm{Id}_n)(AA^\top + \lambda \mathrm{Id}_n)^{-1} = A^\top.$$

Since $A^\top A + \lambda \mathrm{Id}_p$ is invertible, this gives the desired result. □

The solution of these linear systems can be computed using either a direct method such as Cholesky factorization or an iterative method such as a conjugate gradient (which is vastly superior to the vanilla gradient descent scheme).

If $n > p$, then one should use (14.4) while if $n < p$ one should rather use (14.5).

Pseudo-inverse. As $\lambda \to 0$, then $x_\lambda \to x_0$ which is, using (14.2)

$$\operatorname*{argmin}_{Ax=y} \|x\|.$$

If $\ker(A) = \{0\}$ (overdetermined setting), $A^\top A \in \mathbb{R}^{p \times p}$ is an invertible matrix, and $(A^\top A + \lambda \mathrm{Id}_p)^{-1} \to (A^\top A)^{-1}$, so that

$$x_0 = A^+ y \quad \text{where} \quad A^+ \overset{\text{def.}}{=} (A^\top A)^{-1} A^\top.$$

Conversely, if $\ker(A^\top) = \{0\}$, or equivalently $\mathrm{Im}(A) = \mathbb{R}^n$ (undertermined setting) then one has

$$x_0 = A^+ y \quad \text{where} \quad A^+ \overset{\text{def.}}{=} A^\top (AA^\top)^{-1}.$$

In the special case $n = p$ and A is invertible, then both definitions of A^+ coincide, and $A^+ = A^{-1}$. In the general case (where A is neither injective nor surjective), A^+ can be computed using the Singular Values Decomposition (SVD). The matrix A^+ is often called the Moore-Penrose pseudo-inverse.

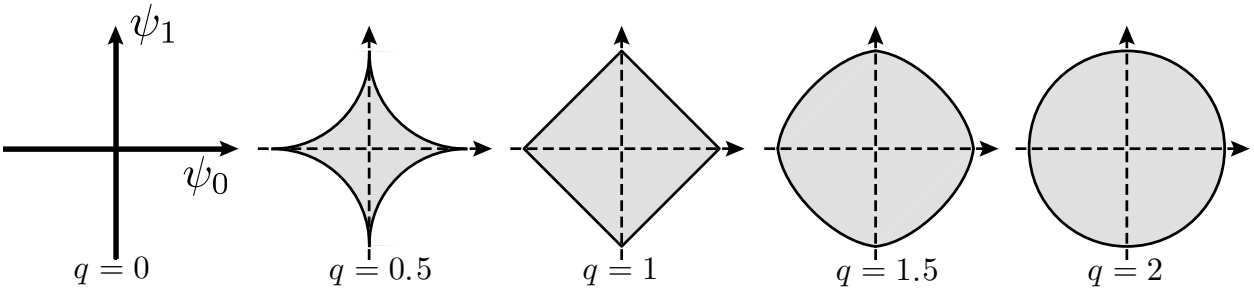

Figure 14.1: ℓ^q balls $\{x \,;\, \sum_k |x_k|^q \leqslant 1\}$ for varying q.

14.1.3 Lasso

The Lasso corresponds to using a ℓ^1 penalty

$$R(x) = \|x\|_1 \overset{\text{def.}}{=} \sum_{k=1}^{p} |x_k|.$$

The underlying idea is that solutions x_λ of a Lasso problem

$$x_\lambda \in \operatorname*{argmin}_{x \in \mathbb{R}^p} f_\lambda(x) = \frac{1}{2}\|Ax - y\|_{\mathbb{R}^n}^2 + \lambda \|x\|_1$$

are sparse, i.e. solutions x_λ (which might be non-unique) have many zero entries. To get some insight about this, Fig. 14.1 display the ℓ^q "balls" which shrink toward the axes as $q \to 0$ (thus enforcing more sparsity) but are non-convex for $q < 1$.

This can serve two purposes: (i) one knows before hand that the solution is expected to be sparse, which is the case for instance in some problems in imaging, (ii) one want to perform model selection by pruning some of the entries in the feature (to have simpler predictor, which can be computed more efficiently at test time, or that can be more interpretable). For typical ML problems though, the performance of the Lasso predictor is usually not better than the one obtained by Ridge.

Minimizing $f(x)$ is still a convex problem, but R is non-smooth, so that one cannot use a gradient descent. Section 14.1.4 shows how to modify the gradient descent to cope with this issue. In general, solutions x_λ cannot be computed in closed form, excepted when the design matrix A is orthogonal.

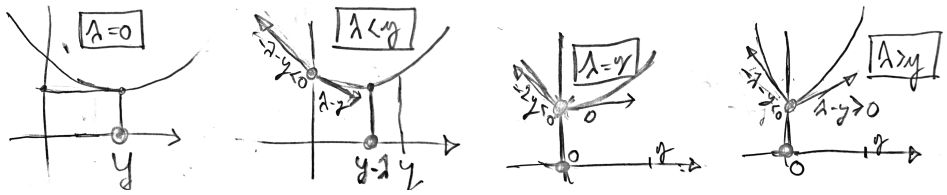

Figure 14.2: Evolution with λ of the function $F(x) \overset{\text{def.}}{=} \frac{1}{2}\|\cdot - y\|^2 + \lambda|\cdot|$.

Proposition 46. *When $n = p$ and $A = \mathrm{Id}_n$, one has*

$$\operatorname*{argmin}_{x \in \mathbb{R}^p} \frac{1}{2}\|x - y\|^2 + \lambda \|x_1\| = S_\lambda(x) \quad \textit{where} \quad S_\lambda(x) = (\operatorname{sign}(x_k)\max(|x_k| - \lambda, 0))_k$$

Proof. One has $f_\lambda(x) = \sum_k \frac{1}{2}(x_k - y_k)^2 + \lambda|x_k|$, so that one needs to find the minimum of the 1-D function $x \in \mathbb{R} \mapsto \frac{1}{2}(x-y)^2 + \lambda|x|$. We can do this minimization "graphically" as shown on Fig. 14.2. For $x > 0$, one has $F'(x) = x - y + \lambda$ wich is 0 at $x = y - \lambda$. The minimum is at $x = y - \lambda$ for $\lambda \leqslant y$, and stays at 0 for all $\lambda > y$. The problem is symmetric with respect to the switch $x \mapsto -x$. □

Here, S_λ is the celebrated soft-thresholding non-linear function.

14.1.4 Iterative Soft Thresholding

We now derive an algorithm using a classical technic of surrogate function minimization. We aim at minimizing

$$f(x) \overset{\text{def.}}{=} \frac{1}{2}\|y - Ax\|^2 + \lambda \|x\|_1$$

and we introduce for any fixed x' the function

$$f_\tau(x,x') \stackrel{\text{def.}}{=} f(x) - \frac{1}{2}\|Ax-Ax'\|^2 + \frac{1}{2\tau}\|x-x'\|^2.$$

We notice that $f_\tau(x,x)=0$ and one the quadratic part of this function reads

$$K(x,x') \stackrel{\text{def.}}{=} -\frac{1}{2}\|Ax-Ax'\|^2 + \frac{1}{2\tau}\|x-x'\| = \frac{1}{2}\langle\left(\frac{1}{\tau}\mathrm{Id}_N - A^\top A\right)(x-x'),\, x-x'\rangle.$$

This quantity $K(x,x')$ is positive if $\lambda_{\max}(A^\top A) \leqslant 1/\tau$ (maximum eigenvalue), i.e. $\tau \leqslant 1/\|A\|_{\text{op}}^2$, where we recall that $\|A\|_{\text{op}} = \sigma_{\max}(A)$ is the operator (algebra) norm. This shows that $f_\tau(x,x')$ is a valid surrogate functional, in the sense that

$$f(x) \leqslant f_\tau(x,x'), \quad f_\tau(x,x')=0, \quad \text{and} \quad f(\cdot) - f_\tau(\cdot,x') \text{ is smooth.}$$

We also note that this majorant $f_\tau(\cdot,x')$ is convex. This leads to define

$$x_{k+1} \stackrel{\text{def.}}{=} \operatorname*{argmin}_x f_\tau(x,x_k) \tag{14.6}$$

which by construction satisfies

$$f(x_{k+1}) \leqslant f(x_k).$$

Proposition 47. *The iterates x_k defined by (14.6) satisfy*

$$x_{k+1} = S_{\lambda\tau}\left(x_k - \tau A^\top(Ax_k - y)\right) \tag{14.7}$$

where $S_\lambda(x) = (s_\lambda(x_m))_m$ where $s_\lambda(r) = \mathrm{sign}(r)\max(|r|-\lambda,0)$ is the soft thresholding operator.

Proof. One has

$$\begin{aligned}
f_\tau(x,x') &= \frac{1}{2}\|Ax-y\|^2 - \frac{1}{2}\|Ax-Ax'\|^2 + \frac{1}{2\tau}\|x-x'\|^2 + \lambda\|x\|_1 \\
&= C + \frac{1}{2}\|Ax\|^2 - \frac{1}{2}\|Ax\|^2 + \frac{1}{2\tau}\|x\|^2 - \langle Ax,\, y\rangle + \langle Ax,\, Ax'\rangle - \frac{1}{\tau}\langle x,\, x'\rangle + \lambda\|x\|_1 \\
&= C + \frac{1}{2\tau}\|x\|^2 + \langle x,\, -A^\top y + AA^\top x' - \frac{1}{\tau}x'\rangle + \lambda\|x\|_1 \\
&= C' + \frac{1}{\tau}\left(\frac{1}{2}\|x-(x'-\tau A^\top(Ax'-y))\|^2 + \tau\lambda\|x\|_1.\right)
\end{aligned}$$

Proposition (46) shows that the minimizer of $f_\tau(x,x')$ is thus indeed $S_{\lambda\tau}(x'-\tau A^\top(Ax'-y))$ as claimed. □

Equation (14.7) defines the iterative soft-thresholding algorithm. It follows from a valid convex surrogate function if $\tau \leqslant 1/\|A\|^2$, but one can actually shows that it converges to a solution of the Lasso as soon as $\tau < 2/\|A\|^2$, which is exactly as for the classical gradient descent.

14.2 Stochastic Optimization

We detail some important stochastic Gradient Descent methods, which enable to perform optimization in the setting where the number of samples n is large and even infinite.

14.2.1 Minimizing Sums and Expectation

A large class of functionals in machine learning can be expressed as minimizing large sums of the form

$$\min_{x\in\mathbb{R}^p} f(x) \stackrel{\text{def.}}{=} \frac{1}{n}\sum_{i=1}^{n} f_i(x) \tag{14.8}$$

or even expectations of the form

$$\min_{x\in\mathbb{R}^p} f(x) \stackrel{\text{def.}}{=} \mathbb{E}_{\mathbf{z}\sim\pi}(f(x,\mathbf{z})) = \int_{\mathcal{Z}} f(x,z)\mathrm{d}\pi(z). \tag{14.9}$$

Problem (14.8) can be seen as a special case of (14.9), when using a discrete empirical uniform measure $\pi = \sum_{i=1}^n \delta_i$ and setting $f(x,i) = f_i(x)$. One can also viewed (14.8) as a discretized "empirical" version of (14.9) when drawing $(z_i)_i$ i.i.d. according to $\mathbf{z}$ and defining $f_i(x) = f(x,z_i)$. In this setup, (14.8) converges to (14.9) as $n \to +\infty$.

A typical example of such a class of problems is empirical risk minimization for linear model, where in these cases

$$f_i(x) = \ell(\langle a_i,\, x\rangle, y_i) \quad \text{and} \quad f(x,z) = \ell(\langle a,\, x\rangle, y) \tag{14.10}$$

for $z = (a,y) \in \mathcal{Z} = (\mathcal{A} = \mathbb{R}^p) \times \mathcal{Y}$ (typically $\mathcal{Y} = \mathbb{R}$ or $\mathcal{Y} = \{-1,+1\}$ for regression and classification), where ℓ is some loss function. We illustrate below the methods on binary logistic classification, where

$$L(s,y) \stackrel{\text{def.}}{=} \log(1+\exp(-sy)). \tag{14.11}$$

But this extends to arbitrary parametric models, and in particular deep neural networks.

While some algorithms (in particular batch gradient descent) are specific to finite sums (14.8), the stochastic methods we detail next work verbatim (with the same convergence guarantees) in the expectation case (14.9). For the sake of simplicity, we however do the exposition for the finite sums case, which is sufficient in the vast majority of cases. But one should keep in mind that n can be arbitrarily large, so it is not acceptable in this setting to use algorithms whose complexity per iteration depend on n.

If the functions $f_i(x)$ are very similar (the extreme case being that they are all equal), then of course there is a gain in using stochastic optimization (since in this case, $\nabla f_i \approx \nabla f$ but ∇f_i is n times cheaper). But in general stochastic optimization methods are notÂ necessarily faster than batch gradient descent. If n is not too large so that one afford the price of doing a few non-stochastic iterations, then deterministic methods can be faster. But if n is so large that one cannot do even a single deterministic iteration, then stochastic methods allow one to have a fine grained scheme by breaking the cost of determinstic iterations in smaller chunks. Another advantage is that they are quite easy to parallelize.

14.2.2 Batch Gradient Descent (BGD)

The usual deterministic (batch) gradient descent (BGD) is studied in details in Section 13.4. Its iterations read

$$x_{k+1} = x_k - \tau_k \nabla f(x_k)$$

and the step size should be chosen as $0 < \tau_{\min} < \tau_k < \tau_{\max} \stackrel{\text{def.}}{=} 2/L$ where L is the Lipschitz constant of the gradient ∇f. In particular, in this deterministic setting, this step size should not go to zero and this ensures quite fast convergence (even linear rates if f is strongly convex).

The computation of the gradient in our setting reads

$$\nabla f(x) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(x) \tag{14.12}$$

so it typically has complexity $O(np)$ if computing ∇f_i has linear complexity in p.

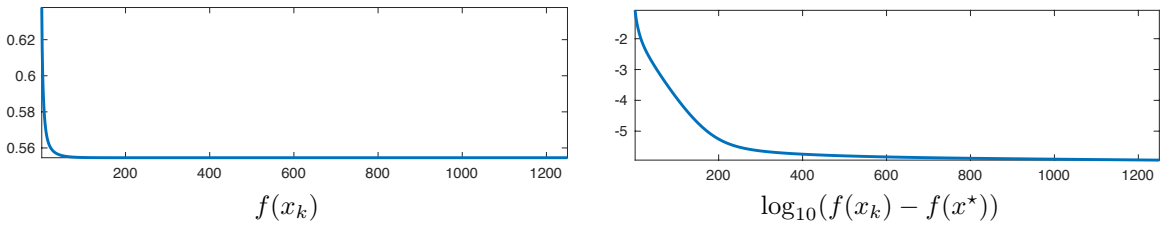

Figure 14.3: Evolution of the error of the BGD for logistic classification.

For ERM-type functions of the form (14.10), one can do the Taylor expansion of f_i

$$\begin{aligned} f_i(x+\varepsilon) &= \ell(\langle a_i,\, x\rangle + \langle a_i,\, \varepsilon\rangle, y_i) = \ell(\langle a_i,\, x\rangle, y_i) + \ell'(\langle a_i,\, x\rangle, y_i)\langle a_i,\, \varepsilon\rangle + o(\|\varepsilon\|) \\ &= f_i(x) + \langle \ell'(\langle a_i,\, x\rangle, y_i)a_i,\, x\rangle + o(\|\varepsilon\|), \end{aligned}$$

where $\ell(y, y') \in \mathbb{R}$ is the derivative with respect to the first variable, i.e. the gradient of the map $y \in \mathbb{R} \mapsto L(y, y') \in \mathbb{R}$. This computation shows that

$$\nabla f_i(x) = \ell'(\langle a_i,\, x\rangle, y_i)a_i. \tag{14.13}$$

For the logistic loss, one has

$$L'(s, y) = -s\frac{e^{-sy}}{1 + e^{-sy}}.$$

14.2.3 Stochastic Gradient Descent (SGD)

For very large n, computing the full gradient ∇f as in (14.12) is prohibitive. The idea of SGD is to trade this exact full gradient by an inexact proxy using a single functional f_i where i is drawn uniformly at random. The main idea that makes this work is that this sampling scheme provides an unbiased estimate of the gradient, in the sense that

$$\mathbb{E}_{\mathbf{i}} \nabla f_{\mathbf{i}}(x) = \nabla f(x) \tag{14.14}$$

where $\mathbf{i}$ is a random variable distributed uniformly in $\{1, \ldots, n\}$.

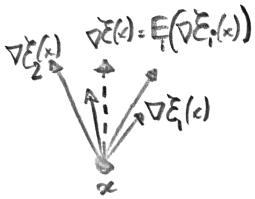

Figure 14.4: Unbiased gradient estimate

Starting from some x_0,the iterations of stochastic gradient descent (SGD) read

$$x_{k+1} = x_k - \tau_k \nabla f_{i(k)}(x_k)$$

where, for each iteration index k, $i(k)$ is drawn uniformly at random in $\{1, \ldots, n\}$. It is important that the iterates x_{k+1} are thus random vectors, and the theoretical analysis of the method thus studies wether this sequence of random vectors converges (in expectation or in probability for instance) toward a deterministic vector (minimizing f), and at which speed.

Note that each step of a batch gradient descent has complexity $O(np)$, while a step of SGD only has complexity $O(p)$. SGD is thus advantageous when n is very large, and one cannot afford to do several passes through the data. In some situation, SGD can provide accurate results even with $k \ll n$, exploiting redundancy between the samples.

A crucial question is the choice of step size schedule τ_k. It must tends to 0 in order to cancel the noise induced on the gradient by the stochastic sampling. But it should not go too fast to zero in order for the method to keep converging.

A typical schedule that ensures both properties is to have asymptotically $\tau_k \sim k^{-1}$ for $k \to +\infty$. We thus propose to use

$$\tau_k \overset{\text{def.}}{=} \frac{\tau_0}{1 + k/k_0} \tag{14.15}$$

Figure 14.5: Schematic view of SGD iterates

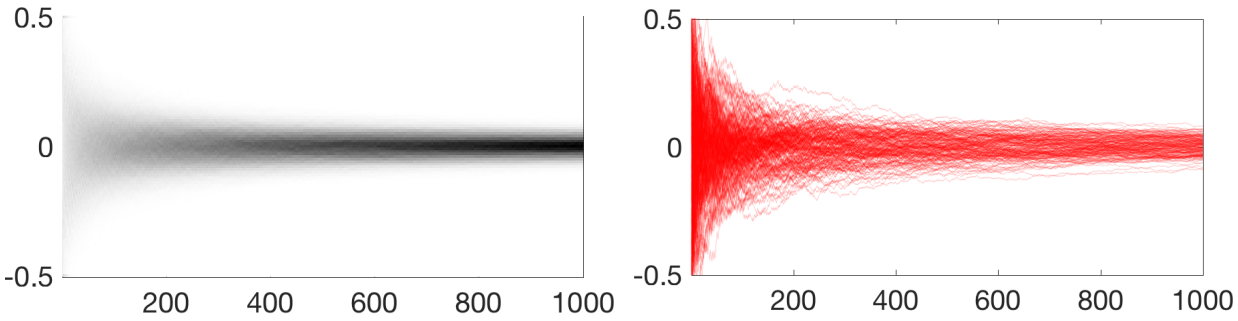

Figure 14.6: Display of a large number of trajectories $k \mapsto x_k \in \mathbb{R}$ generated by several runs of SGD. On the top row, each curve is a trajectory, and the bottom row displays the corresponding density.

where k_0 indicates roughly the number of iterations serving as a "warmup" phase.

Figure 14.6 shows a simple 1-D example to minimize $f_1(x) + f_2(x)$ for $x \in \mathbb{R}$ and $f_1(x) = (x-1)^2$ and $f_2(x) = (x+1)^2$. One can see how the density of the distribution of x_k progressively clusters around the minimizer $x^\star = 0$. Here the distribution of x_0 is uniform on $[-1/2, 1/2]$.

The following theorem shows the convergence in expectation with a $1/\sqrt{k}$ rate on the objective.

Theorem 24. *We assume f is μ-strongly convex as defined in ($\mathcal{S}_\mu$) (i.e. $\mathrm{Id}_p \preceq \partial^2 f(x)$ if f is $\mathcal{C}^2$), and is such that $\|\nabla f_i(x)\|^2 \leqslant C^2$. For the step size choice $\tau_k = \frac{1}{\mu(k+1)}$, one has*

$$\mathbb{E}(\|x_k - x^\star\|^2) \leqslant \frac{R}{k+1} \quad \textit{where} \quad R = \max(\|x_0 - x^\star\|, C^2/\mu^2), \tag{14.16}$$

where $\mathbb{E}$ indicates an expectation with respect to the i.i.d. sampling performed at each iteration.

Proof. By strong convexity, one has

$$\begin{aligned} f(x^\star) - f(x_k) &\geqslant \langle \nabla f(x_k),\, x^\star - x_k \rangle + \frac{\mu}{2}\|x_k - x^\star\|^2 \\ f(x_k) - f(x^\star) &\geqslant \langle \nabla f(x^\star),\, x_k - x^\star \rangle + \frac{\mu}{2}\|x_k - x^\star\|^2. \end{aligned}$$

Summing these two inequalities and using $\nabla f(x^\star) = 0$ leads to

$$\langle \nabla f(x_k) - \nabla f(x^\star),\, x_k - x^\star \rangle = \langle \nabla f(x_k),\, x_k - x^\star \rangle \geqslant \mu \|x_k - x^\star\|^2. \tag{14.17}$$

Considering only the expectation with respect to the ransom sample of $i(k) \sim \mathbf{i}_k$, one has

$$\begin{aligned} \mathbb{E}_{\mathbf{i}_k}(\|x_{k+1} - x^\star\|^2) &= \mathbb{E}_{\mathbf{i}_k}(\|x_k - \tau_k \nabla f_{\mathbf{i}_k}(x_k) - x^\star\|^2) \\ &= \|x_k - x^\star\|^2 + 2\tau_k \langle \mathbb{E}_{\mathbf{i}_k}(\nabla f_{\mathbf{i}_k}(x_k)),\, x^\star - x_k \rangle + \tau_k^2 \mathbb{E}_{\mathbf{i}_k}(\|\nabla f_{\mathbf{i}_k}(x_k)\|^2) \\ &\leqslant \|x_k - x^\star\|^2 + 2\tau_k \langle \nabla f(x_k)),\, x^\star - x_k \rangle + \tau_k^2 C^2 \end{aligned}$$

where we used the fact (14.14) that the gradient is unbiased. Taking now the full expectation with respect to all the other previous iterates, and using (14.17) one obtains

$$\mathbb{E}(\|x_{k+1} - x^\star\|^2) \leqslant \mathbb{E}(\|x_k - x^\star\|^2) - 2\mu\tau_k \mathbb{E}(\|x_k - x^\star\|^2) + \tau_k^2 C^2 = (1 - 2\mu\tau_k)\mathbb{E}(\|x_k - x^\star\|^2) + \tau_k^2 C^2. \tag{14.18}$$

We show by recursion that the bound (14.16) holds. We denote $\varepsilon_k \overset{\text{def.}}{=} \mathbb{E}(\|x_k - x^\star\|^2)$. Indeed, for $k = 0$, this it is true that

$$\varepsilon_0 \leqslant \frac{\max(\|x_0 - x^\star\|, C^2/\mu^2)}{1} = \frac{R}{1}.$$

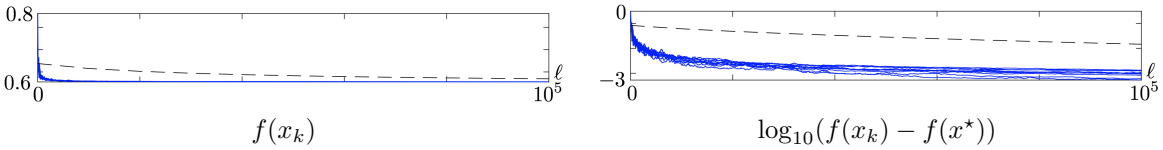

Figure 14.7: Evolution of the error of the SGD for logistic classification (dashed line shows BGD).

We now assume that $\varepsilon_k \leqslant \frac{R}{k+1}$. Using (14.18) in the case of $\tau_k = \frac{1}{\mu(k+1)}$, one has, denoting $m = k+1$

$$\begin{aligned}\varepsilon_{k+1} &\leqslant (1 - 2\mu\tau_k)\varepsilon_k + \tau_k^2 C^2 = \left(1 - \frac{2}{m}\right)\varepsilon_k + \frac{C^2}{(\mu m)^2} \\ &\leqslant \left(1 - \frac{2}{m}\right)\frac{R}{m} + \frac{R}{m^2} = \left(\frac{1}{m} - \frac{1}{m^2}\right)R = \frac{m-1}{m^2}R = \frac{m^2-1}{m^2}\frac{1}{m+1}R \leqslant \frac{R}{m+1}\end{aligned}$$

□

A weakness of SGD (as well as the SGA scheme studied next) is that it only weakly benefit from strong convexity of f. This is in sharp contrast with BGD, which enjoy a fast linear rate for strongly convex functionals, see Theorem 23.

Figure 14.7 displays the evolution of the energy $f(x_k)$. It overlays on top (black dashed curve) the convergence of the batch gradient descent, with a careful scaling of the number of iteration to account for the fact that the complexity of a batch iteration is n times larger.

14.2.4 Stochastic Gradient Descent with Averaging (SGA)

Stochastic gradient descent is slow because of the fast decay of τ_k toward zero. To improve somehow the convergence speed, it is possible to average the past iterate, i.e. run a "classical" SGD on auxiliary variables $(\tilde{x}_k)_k$

$$\tilde{x}^{(\ell+1)} = \tilde{x}_k - \tau_k \nabla f_{i(k)}(\tilde{x}_k)$$

and output as estimated weight vector the Cesaro average

$$x_k \overset{\text{def.}}{=} \frac{1}{k}\sum_{\ell=1}^{k} \tilde{x}_\ell.$$

This defines the Stochastic Gradient Descent with Averaging (SGA) algorithm.

Note that it is possible to avoid explicitly storing all the iterates by simply updating a running average as follow

$$x_{k+1} = \frac{1}{k}\tilde{x}_k + \frac{k-1}{k}x_k.$$

In this case, a typical choice of decay is rather of the form

$$\tau_k \overset{\text{def.}}{=} \frac{\tau_0}{1 + \sqrt{k/k_0}}.$$

Notice that the step size now goes much slower to 0, at rate $k^{-1/2}$.

Typically, because the averaging stabilizes the iterates, the choice of (k_0, τ_0) is less important than for SGD.

Bach proves that for logistic classification, it leads to a faster convergence (the constant involved are smaller) than SGD, since on contrast to SGD, SGA is adaptive to the local strong convexity of E.

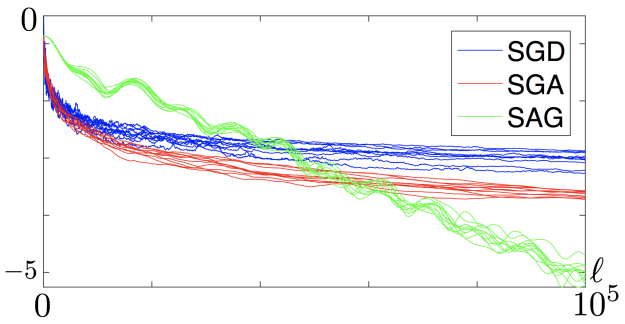

Figure 14.8: Evolution of $\log_{10}(f(x_k) - f(x^\star))$ for SGD, SGA and SAG.

14.2.5 Stochastic Averaged Gradient Descent (SAG)

For problem size n where the dataset (of size $n \times p$) can fully fit into memory, it is possible to further improve the SGA method by bookkeeping the previous gradients. This gives rise to the Stochastic Averaged Gradient Descent (SAG) algorithm.

We store all the previously computed gradients in $(G^i)_{i=1}^n$, which necessitates $O(n \times p)$ memory. The iterates are defined by using a proxy g for the batch gradient, which is progressively enhanced during the iterates.

The algorithm reads

$$x_{k+1} = x_k - \tau g \quad \text{where} \quad \left\{ \begin{array}{l} h \leftarrow \nabla f_{i(k)}(\tilde{x}_k), \\ g \leftarrow g - G^{i(k)} + h, \\ G^{i(k)} \leftarrow h. \end{array} \right.$$

Note that in contrast to SGD and SGA, this method uses a fixed step size τ. Similarly to the BGD, in order to ensure convergence, the step size τ should be of the order of $1/L$ where L is the Lipschitz constant of f.

This algorithm improves over SGA and SGD since it has a convergence rate of $O(1/k)$ as does BGD. Furthermore, in the presence of strong convexity (for instance when X is injective for logistic classification), it has a linear convergence rate, i.e.

$$\mathbb{E}(f(x_k)) - f(x^\star) = O\left(\rho^k\right),$$

for some $0 < \rho < 1$.

Note that this improvement over SGD and SGA is made possible only because SAG explicitly uses the fact that n is finite (while SGD and SGA can be extended to infinite n and more general minimization of expectations (14.9)).

Figure 14.8 shows a comparison of SGD, SGA and SAG.

14.3 Automatic Differentiation

The main computational bottleneck of gradient descent methods (batch or stochastic) is the computation of gradients $\nabla f(x)$. For simple functionals, such as those encountered in ERM for linear models, and also for MLP with a single hidden layer, it is possible to compute these gradients in closed form, and that the main computational burden is the evaluation of matrix-vector products. For more complicated functionals (such as those involving deep networks), computing the formula for the gradient quickly becomes cumbersome. Even worse: computing these gradients using the usual chain rule formula is sub-optimal. We presents methods to compute recursively in an optimal manner these gradients. The purpose of this approach is to automatize this computational step.

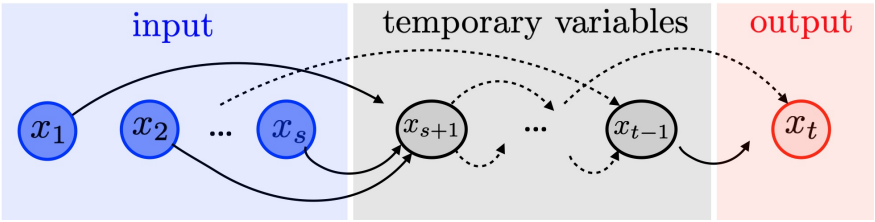

Figure 14.9: A computational graph.

14.3.1 Finite Differences and Symbolic Calculus

We consider $f : \mathbb{R}^p \to \mathbb{R}$ and want to derive a method to evaluate $\nabla f : \mathbb{R}^p \mapsto \mathbb{R}^p$. Approximating this vector field using finite differences, i.e. introducing $\varepsilon > 0$ small enough and computing

$$\frac{1}{\varepsilon}(f(x+\varepsilon\delta_1) - f(x), \ldots, f(x+\varepsilon\delta_p) - f(x))^\top \approx \nabla f(x)$$

requires $p+1$ evaluations of f, where we denoted $\delta_k = (0, \ldots, 0, 1, 0, \ldots, 0)$ where the 1 is at index k. For a large p, this is prohibitive. The method we describe in this section (the so-called reverse mode automatic differentiation) has in most cases a cost proportional to a single evaluation of f. This type of method is similar to symbolic calculus in the sense that it provides (up to machine precision) exact gradient computation. But symbolic calculus does not takes into account the underlying algorithm which compute the function, while automatic differentiation factorizes the computation of the derivative according to an efficient algorithm.

14.3.2 Computational Graphs

We consider a generic function $f(x)$ where $x = (x_1, \ldots, x_s)$ are the input variables. We assume that f is implemented in an algorithm, with intermediate variable $(x_{s+1}, \ldots, x_t)$ where t is the total number of variables. The output is x_t, and we thus denote $x_t = f(x)$ this function. We denote $x_k \in \mathbb{R}^{n_k}$ the dimensionality of the variables. The goal is to compute the derivatives $\frac{\partial f(x)}{\partial x_k} \in \mathbb{R}^{n_t \times n_k}$ for $k = 1, \ldots, s$. For the sake of simplicity, one can assume in what follows that $n_k = 1$ so that all the involved quantities are scalar (but if this is not the case, beware that the order of multiplication of the matrices of course matters).

A numerical algorithm can be represented as a succession of functions of the form

$$\forall\, k = s+1, \ldots, t, \quad x_k = f_k(x_1, \ldots, x_{k-1})$$

where f_k is a function which only depends on the previous variables, see Fig. 14.9. One can represent this algorithm using a directed acyclic graph (DAG), linking the variables involved in f_k to x_k. The node of this graph are thus conveniently ordered by their indexing, and the directed edges only link a variable to another one with a strictly larger index. The evaluation of $f(x)$ thus corresponds to a forward traversal of this graph. Note that the goal of automatic differentiation is not to define an efficient computational graph, it is up to the user to provide this graph. Computing an efficient graph associated to a mathematical formula is a complicated combinatorial problem, which still has to be solved by the user. Automatic differentiation thus leverage the availability of an efficient graph to provide an efficient algorithm to evaluate derivatives.

14.3.3 Forward Mode of Automatic Differentiation

The forward mode correspond to the usual way of computing differentials. It compute the derivative $\frac{\partial x_k}{\partial x_1}$ of all variables x_k with respect to x_1. One then needs to repeat this method p times to compute all the derivative with respect to $x_1, x_2, \ldots, x_p$ (we only write thing for the first variable, the method being of course the same with respect to the other ones).

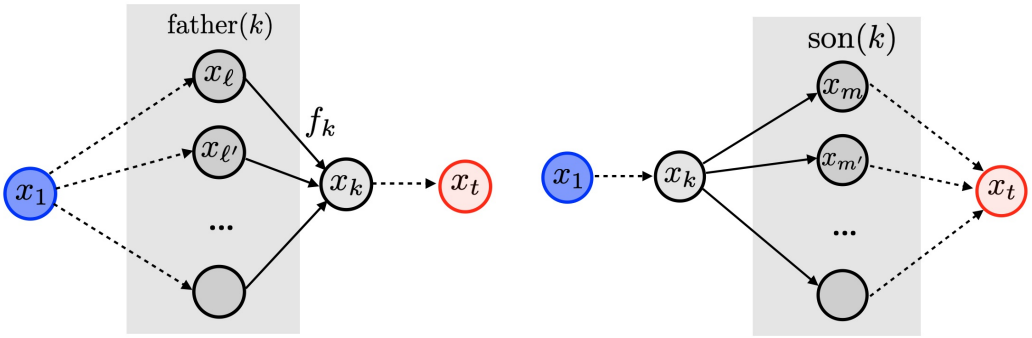

Figure 14.10: Relation between the variable for the forward (left) and backward (right) modes.

The method initialize the derivative of the input nodes

$$\frac{\partial x_1}{\partial x_1} = \mathrm{Id}_{n_1 \times n_1}, \quad \frac{\partial x_2}{\partial x_1} = 0_{n_2 \times n_1}, \ldots, \quad \frac{\partial x_s}{\partial x_1} = 0_{n_s \times n_1},$$

(and thus 1 and 0's for scalar variables), and then iteratively make use of the following recursion formula

$$\forall\, k = s+1, \ldots, t, \quad \frac{\partial x_k}{\partial x_1} = \sum_{\ell \in \mathrm{parent}(k)} \left[\frac{\partial x_k}{\partial x_\ell}\right] \times \frac{\partial x_\ell}{\partial x_1} = \sum_{\ell \in \mathrm{parent}(k)} \frac{\partial f_k}{\partial x_\ell}(x_1, \ldots, x_{k-1}) \times \frac{\partial x_\ell}{\partial x_1}.$$

The notation "parent(k)" denotes the nodes $\ell < k$ of the graph that are connected to k, see Figure 14.10, left. Here the quantities being computed (i.e. stored in computer variables) are the derivatives $\frac{\partial x_\ell}{\partial x_1}$, and $\times$ denotes in full generality matrix-matrix multiplications. We have put in $[\ldots]$ an informal notation, since here $\frac{\partial x_k}{\partial x_\ell}$ should be interpreted not as a numerical variable but needs to be interpreted as derivative of the function f_k, which can be evaluated on the fly (we assume that the derivative of the function involved are accessible in closed form).

Assuming all the involved functions $\frac{\partial f_k}{\partial x_k}$ have the same complexity (which is likely to be the case if all the n_k are for instance scalar or have the same dimension), and that the number of parent node is bounded, one sees that the complexity of this scheme is p times the complexity of the evaluation of f (since this needs to be repeated p times for $\frac{\partial}{\partial x_1}, \ldots, \frac{\partial}{\partial x_p}$). For a large p, this is prohibitive.

Simple example. We consider the fonction

$$f(x, y) = y \log(x) + \sqrt{y \log(x)} \tag{14.19}$$

whose computational graph is displayed on Figure 14.11. The iterations of the forward mode to compute the derivative with respect to x read

$$\begin{aligned}
&\frac{\partial x}{\partial x} = 1, \quad \frac{\partial y}{\partial x} = 0 \\
&\frac{\partial a}{\partial x} = \left[\frac{\partial a}{\partial x}\right] \frac{\partial x}{\partial x} = \frac{1}{x} \frac{\partial x}{\partial x} && \{x \mapsto a = \log(x)\} \\
&\frac{\partial b}{\partial x} = \left[\frac{\partial b}{\partial a}\right] \frac{\partial a}{\partial x} + \left[\frac{\partial b}{\partial y}\right] \frac{\partial y}{\partial x} = y \frac{\partial a}{\partial x} + 0 && \{(y, a) \mapsto b = ya\} \\
&\frac{\partial c}{\partial x} = \left[\frac{\partial c}{\partial b}\right] \frac{\partial b}{\partial x} = \frac{1}{2\sqrt{b}} \frac{\partial b}{\partial x} && \{b \mapsto c = \sqrt{b}\} \\
&\frac{\partial f}{\partial x} = \left[\frac{\partial f}{\partial b}\right] \frac{\partial b}{\partial x} + \left[\frac{\partial f}{\partial c}\right] \frac{\partial c}{\partial x} = 1 \frac{\partial b}{\partial x} + 1 \frac{\partial c}{\partial x} && \{(b, c) \mapsto f = b + c\}
\end{aligned}$$

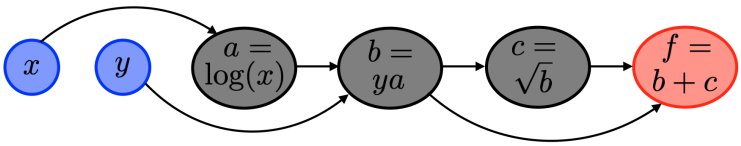

Figure 14.11: Example of a simple computational graph.

One needs to run another forward pass to compute the derivative with respect to y

$$\begin{aligned}
&\frac{\partial x}{\partial y} = 0, \quad \frac{\partial y}{\partial y} = 1 \\
&\frac{\partial a}{\partial y} = \left[\frac{\partial a}{\partial x}\right]\frac{\partial x}{\partial y} = 0 && \{x \mapsto a = \log(x)\} \\
&\frac{\partial b}{\partial y} = \left[\frac{\partial b}{\partial a}\right]\frac{\partial a}{\partial y} + \left[\frac{\partial b}{\partial y}\right]\frac{\partial y}{\partial y} = 0 + a\frac{\partial y}{\partial y} && \{(y,a) \mapsto b = ya\} \\
&\frac{\partial c}{\partial y} = \left[\frac{\partial c}{\partial b}\right]\frac{\partial b}{\partial y} = \frac{1}{2\sqrt{b}}\frac{\partial b}{\partial y} && \{b \mapsto c = \sqrt{b}\} \\
&\frac{\partial f}{\partial y} = \left[\frac{\partial f}{\partial b}\right]\frac{\partial b}{\partial y} + \left[\frac{\partial f}{\partial c}\right]\frac{\partial c}{\partial y} = 1\frac{\partial b}{\partial y} + 1\frac{\partial c}{\partial y} && \{(b,c) \mapsto f = b + c\}
\end{aligned}$$

Dual numbers. A convenient way to implement this forward pass is to make use of so called "dual number", which is an algebra over the real where the number have the form $x + \varepsilon x'$ where ε is a symbol obeying the rule that $\varepsilon^2 = 0$. Here $(x, x') \in \mathbb{R}^2$ and x' is intended to store a derivative with respect to some input variable. These number thus obeys the following arithmetic operations

$$(x + \varepsilon x')(y + \varepsilon y') = xy + \varepsilon(xy' + yx') \quad \text{and} \quad \frac{1}{x + \varepsilon x'} = \frac{1}{x} - \varepsilon\frac{x'}{x^2}.$$

If f is a polynomial or a rational function, from these rules one has that

$$f(x + \varepsilon) = f(x) + \varepsilon f'(x).$$

For a more general basic function f, one needs to overload it so that

$$f(x + \varepsilon x') \stackrel{\text{def.}}{=} f(x) + \varepsilon f'(x)x'.$$

Using this definition, one has that

$$(f \circ g)(x + \varepsilon) = f(g(x)) + \varepsilon f'(g(x))g'(x)$$

which corresponds to the usual chain rule. More generally, if $f(x_1, \ldots, x_s)$ is a function implemented using these overloaded basic functions, one has

$$f(x_1 + \varepsilon, x_2, \ldots, x_s) = f(x_1, \ldots, x_s) + \varepsilon\frac{\partial f}{\partial x_1}(x_1, \ldots, x_s)$$

and this evaluation is equivalent to applying the forward mode of automatic differentiation to compute $\frac{\partial f}{\partial x_1}(x_1, \ldots, x_s)$ (and similarly for the other variables).

14.3.4 Reverse Mode of Automatic Differentiation

Instead of evaluating the differentials $\frac{\partial x_k}{\partial x_1}$ which is problematic for a large p, the reverse mode evaluates the differentials $\frac{\partial x_t}{\partial x_k}$, i.e. it computes the derivative of the output node with respect to the all the inner nodes.

The method initialize the derivative of the final node

$$\frac{\partial x_t}{\partial x_t} = \mathrm{Id}_{n_t \times n_t},$$

and then iteratively makes use, from the last node to the first, of the following recursion formula

$$\forall\, k = t-1, t-2, \ldots, 1, \quad \frac{\partial x_t}{\partial x_k} = \sum_{m \in \mathrm{son}(k)} \frac{\partial x_t}{\partial x_m} \times \left[\frac{\partial x_m}{\partial x_k}\right] = \sum_{m \in \mathrm{son}(k)} \frac{\partial x_t}{\partial x_m} \times \frac{\partial f_m(x_1, \ldots, x_m)}{\partial x_k}.$$

The notation "parent(k)" denotes the nodes $\ell < k$ of the graph that are connected to k, see Figure 14.10, right.

Back-propagation. In the special case where $x_t \in \mathbb{R}$, then $\frac{\partial x_t}{\partial x_k} = [\nabla_{x_k} f(x)]^\top \in \mathbb{R}^{1 \times n_k}$ and one can write the recursion on the gradient vector as follow

$$\forall\, k = t-1, t-2, \ldots, 1, \quad \nabla_{x_k} f(x) = \sum_{m \in \mathrm{son}(k)} \left(\frac{\partial f_m(x_1, \ldots, x_m)}{\partial x_k}\right)^\top (\nabla_{x_m} f(x)).$$

where $\left(\frac{\partial f_m(x_1,\ldots,x_m)}{\partial x_k}\right)^\top \in \mathbb{R}^{n_k \times n_m}$ is the adjoint of the Jacobian of f_m. This form of recursion using adjoint is often referred to as "back-propagation", and is the most frequent setting in applications to ML.

In general, when $n_t = 1$, the backward is the optimal way to compute the gradient of a function. Its drawback is that it necessitate the pre-computation of all the intermediate variables $(x_k)_{k=p}^t$, which can be prohibitive in term of memory usage when t is large. There exists check-pointing method to alleviate this issue, but it is out of the scope of this course.

Simple example. We consider once again the fonction $f(x)$ of (14.19), the iterations of the reverse mode read

$$\begin{aligned}
&\frac{\partial f}{\partial f} = 1 \\
&\frac{\partial f}{\partial c} = \frac{\partial f}{\partial f}\left[\frac{\partial f}{\partial c}\right] = \frac{\partial f}{\partial f} 1 && \{c \mapsto f = b + c\} \\
&\frac{\partial f}{\partial b} = \frac{\partial f}{\partial c}\left[\frac{\partial c}{\partial b}\right] + \frac{\partial f}{\partial f}\left[\frac{\partial f}{\partial b}\right] = \frac{\partial f}{\partial c}\frac{1}{2\sqrt{b}} + \frac{\partial f}{\partial f} 1 && \{b \mapsto c = \sqrt{b}, b \mapsto f = b + c\} \\
&\frac{\partial f}{\partial a} = \frac{\partial f}{\partial b}\left[\frac{\partial b}{\partial a}\right] = \frac{\partial f}{\partial b} y && \{a \mapsto b = ya\} \\
&\frac{\partial f}{\partial y} = \frac{\partial f}{\partial b}\left[\frac{\partial b}{\partial y}\right] = \frac{\partial f}{\partial b} a && \{y \mapsto b = ya\} \\
&\frac{\partial f}{\partial x} = \frac{\partial f}{\partial a}\left[\frac{\partial a}{\partial x}\right] = \frac{\partial f}{\partial a}\frac{1}{x} && \{x \mapsto a = \log(x)\}
\end{aligned}$$

The advantage of the reverse mode is that a single traversal of the computational graph allows to compute both derivatives with respect to x, y, while the forward more necessitates two passes.

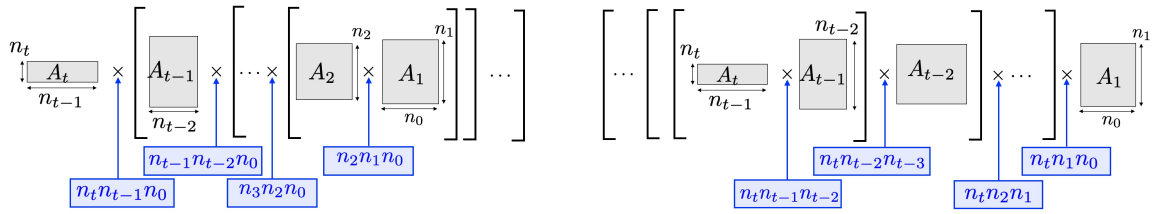

Figure 14.12: Complexity of forward (left) and backward (right) modes for composition of functions.

14.3.5 Feed-forward Compositions

The simplest computational graphs are purely feedforward, and corresponds to the computation of

$$f = f_t \circ f_{t-1} \circ \ldots \circ f_2 \circ f_1 \tag{14.20}$$

for functions $f_k : \mathbb{R}^{n_{k-1}} \to \mathbb{R}^{n_k}$.

The forward function evaluation algorithm initializes $x_0 = x \in \mathbb{R}^{n_0}$ and then computes

$$\forall\, k = 1, \ldots, t, \quad x_k = f_k(x_{k-1})$$

where at the output, one retrieves $f(x) = x_t$.

Denoting $A_k \overset{\text{def.}}{=} \partial f_k(x_{k-1}) \in \mathbb{R}^{n_k \times n_{k-1}}$ the Jacobian, one has

$$\partial f(x) = A_t \times A_{t-1} \times \ldots A_2 \times A_1.$$

The forward (resp. backward) mode corresponds to the computation of the product of the Jacobian from right to left (resp. left to right)

$$\begin{aligned}
\partial f(x) &= A_t \times (A_{t-1} \times (\ldots \times (A_3 \times (A_2 \times A_1)))), \\
\partial f(x) &= ((((A_t \times A_{t-1}) \times A_{t-2}) \times \ldots) \times A_2) \times A_1.
\end{aligned}$$

We note that the computation of the product $A \times B$ of $A \in \mathbb{R}^{n \times p}$ with $B \in \mathbb{R}^{p \times q}$ necessitates npq operations. As shown on Figure 14.12, the complexity of the forward and backward modes are

$$n_0 \sum_{k=1}^{t-1} n_k n_{k+1} \quad \text{and} \quad n_t \sum_{k=0}^{t-2} n_k n_{k+1}$$

So if $n_t \ll n_0$ (which is the typical case in ML scenario where $n_t = 1$) then the backward mode is cheaper.

14.3.6 Feed-forward Architecture

We can generalize the previous example to account for feed-forward architectures, such as neural networks, which are of the form

$$\forall\, k = 1, \ldots, t, \quad x_k = f_k(x_{k-1}, \theta_{k-1}) \tag{14.21}$$

where θ_{k-1} is a vector of parameters and $x_0 \in \mathbb{R}^{n_0}$ is given. The function to minimize has the form

$$f(\theta) \overset{\text{def.}}{=} L(x_t) \tag{14.22}$$

where $L : \mathbb{R}^{n_t} \to \mathbb{R}$ is some loss function (for instance a least square or logistic prediction risk) and $\theta = (\theta_k)_{k=0}^{t-1}$. Figure 14.13, top, displays the associated computational graph.

One can use the reverse mode automatic differentiation to compute the gradient of f by computing successively the gradient with respect to all (x_k, θ_k). One initializes

$$\nabla_{x_t} f = \nabla L(x_t)$$

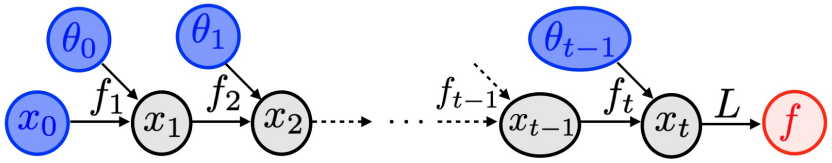

Figure 14.13: Computational graph for a feedforward architecture.

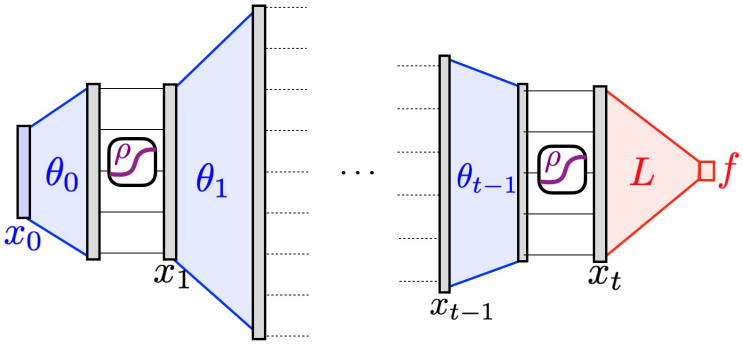

Figure 14.14: Multi-layer perceptron parameterization.

and then recurse from $k = t-1$ to 0

$$z_{k-1} = [\partial_x f_k(x_{k-1}, \theta_{k-1})]^\top z_k \quad \text{and} \quad \nabla_{\theta_{k-1}} f = [\partial_\theta f_k(x_{k-1}, \theta_{k-1})]^\top (\nabla_{x_k} f) \tag{14.23}$$

where we denoted $z_k \overset{\text{def.}}{=} \nabla_{x_k} f(\theta)$ the gradient with respect to x_k.

Multilayers perceptron. For instance, feedforward deep network (fully connected for simplicity) corresponds to using

$$\forall\, x_{k-1} \in \mathbb{R}^{n_{k-1}}, \quad f_k(x_{k-1}, \theta_{k-1}) = \rho(\theta_{k-1} x_{k-1}) \tag{14.24}$$

where $\theta_{k-1} \in \mathbb{R}^{n_k \times n_{k-1}}$ are the neuron's weights and ρ a fixed pointwise linearity, see Figure 14.14. One has, for a vector $z_k \in \mathbb{R}^{n_k}$ (typically equal to $\nabla_{x_k} f$)

$$\begin{cases} [\partial_x f_k(x_{k-1}, \theta_{k-1})]^\top (z_k) = \theta_{k-1}^\top w_k z_k, \\ [\partial_\theta f_k(x_{k-1}, \theta_{k-1})]^\top (z_k) = w_k x_{k-1}^\top \end{cases} \quad \text{where} \quad w_k \overset{\text{def.}}{=} \operatorname{diag}(\rho'(\theta_{k-1} x_{k-1})).$$

Link with adjoint state method. One can interpret (14.21) as a time discretization of a continuous ODE. One imposes that the dimension $n_k = n$ is fixed, and denotes $x(t) \in \mathbb{R}^n$ a continuous time evolution, so that $x_k \to x(k\tau)$ when $k \to +\infty$ and $k\tau \to t$. Imposing then the structure

$$f_k(x_{k-1}, \theta_{k-1}) = x_{k-1} + \tau u(x_{k-1}, \theta_{k-1}, k\tau) \tag{14.25}$$

where $u(x, \theta, t) \in \mathbb{R}^n$ is a parameterized vector field, as $\tau \to 0$, one obtains the non-linear ODE

$$\dot{x}(t) = u(x(t), \theta(t), t) \tag{14.26}$$

with $x(t=0) = x_0$.

Denoting $z(t) = \nabla_{x(t)} f(\theta)$ the "adjoint" vector field, the discrete equations (14.28) becomes the so-called adjoint equations, which is a linear ODE

$$\dot{z}(t) = -[\partial_x u(x(t), \theta(t), t)]^\top z(t) \quad \text{and} \quad \nabla_{\theta(t)} f(\theta) = [\partial_\theta u(x(t), \theta(t), t)]^\top z(t).$$

Note that the correct normalization is $\frac{1}{\tau} \nabla_{\theta_{k-1}} f \to \nabla_{\theta(t)} f(\theta)$

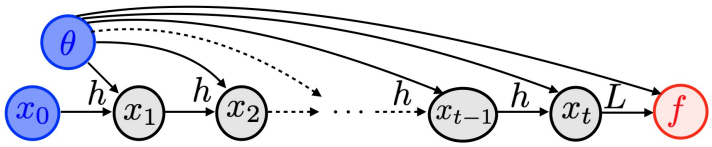

Figure 14.15: Computational graph for a recurrent architecture.

14.3.7 Recurrent Architectures

Parametric recurrent functions are obtained by using the same parameter $\theta = \theta_k$ and $f_k = h$ recursively in (14.24), so that

$$\forall\, k = 1, \ldots, t, \quad x_k = h(x_{k-1}, \theta). \tag{14.27}$$

We consider a real valued function of the form

$$f(\theta) = L(x_t, \theta)$$

so that here the final loss depends on θ (which is thus more general than (14.22)). Figure 14.15, bottom, displays the associated computational graph.

The back-propagation then operates as

$$\nabla_{x_{k-1}} f = [\partial_x h(x_{k-1}, \theta)]^\top \nabla_{x_k} f \quad \text{and} \quad \nabla_\theta f = \nabla_\theta L(x_t, \theta) + \sum_k [\partial_\theta h(x_{k-1}, \theta)]^\top \nabla_{x_k} f. \tag{14.28}$$

Similarly, writing $h(x, \theta) = x + \tau u(x, \theta)$, letting $(k, k\tau) \to (+\infty, t)$, one obtains the forward non-linear ODE with a time-stationary vector field

$$\dot{x}(t) = u(x(t), \theta)$$

and the following linear backward adjoint equation, for $f(\theta) = L(x(T), \theta)$

$$\dot{z}(t) = -[\partial_x u(x(t), \theta)]^\top z(t) \quad \text{and} \quad \nabla_\theta f(\theta) = \nabla_\theta L(x(T), \theta) + \int_0^T [\partial_\theta f(x(t), \theta)]^\top z(t) \mathrm{d}t. \tag{14.29}$$

with $z(0) = \nabla_x L(x_t, \theta)$.

Residual recurrent networks. A recurrent network is defined using

$$h(x, \theta) = x + W_2^\top \rho(W_1 x)$$

as displayed on Figure 14.16, where $\theta = (W_1, W_2) \in (\mathbb{R}^{n \times q})^2$ are the weights and ρ is a pointwise non-linearity. The number q of hidden neurons can be increased to approximate more complex functions. In the special case where $W_2 = -\tau W_1$, and $\rho = \psi'$, then this is a special case of an argmin layer (14.31) to minimize the function $\mathcal{E}(x, \theta) = \psi(W_1 x)$ using gradient descent, where $\psi(u) = \sum_i \psi(u_i)$ is a separable function. The Jacobians $\partial_\theta h$ and $\partial_x h$ are computed similarly to (14.29).

Mitigating memory requirement. The main issue of applying this backpropagation method to compute $\nabla f(\theta)$ is that it requires a large memory to store all the iterates $(x_k)_{k=0}^t$. A workaround is to use checkpointing, which stores some of these intermediate results and re-run partially the forward algorithm to reconstruct missing values during the backward pass. Clever hierarchical method perform this recursively in order to only require $\log(t)$ stored values and a $\log(t)$ increase on the numerical complexity.

In some situation, it is possible to avoid the storage of the forward result, if one assume that the algorithm can be run backward. This means that there exists some functions g_k so that

$$x_k = g_k(x_{k+1}, \ldots, x_t).$$

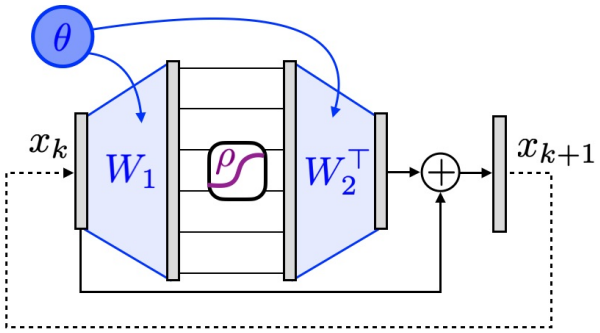

Figure 14.16: Recurrent residual perceptron parameterization.

In practice, this function typically also depends on a few extra variables, in particular on the input values $(x_0, \ldots, x_s)$.

An example of this situation is when one can split the (continuous time) variable as $x(t) = (r(t), s(t))$ and the vector field u in the continuous ODE (14.26) has a symplectic structure of the form $u((r,s),\theta,t) = (F(s,\theta,t), G(r,\theta,t))$. One can then use a leapfrog integration scheme, which defines

$$r_{k+1} = r_k + \tau F(s_k, \theta_k, \tau k) \quad \text{and} \quad s_{k+1} = s_k + \tau G(r_{k+1}, \theta_{k+1/2}, \tau(k+1/2)).$$

One can reverse these equation exactly as

$$s_k = s_{k+1} - \tau G(r_{k+1}, \theta_{k+1/2}, \tau(k+1/2)). \quad \text{and} \quad r_k = r_{k+1} - \tau F(s_k, \theta_k, \tau k).$$

Fixed point maps In some applications (some of which are detailed below), the iterations x_k converges to some $x^\star(\theta)$ which is thus a fixed point

$$x^\star(\theta) = h(x^\star(\theta), \theta).$$

Instead of applying the back-propagation to compute the gradient of $f(\theta) = L(x_t, \theta)$, one can thus apply the implicit function theorem to compute the gradient of $f^\star(\theta) = L(x^\star(\theta), \theta)$. Indeed, one has

$$\nabla f^\star(\theta) = [\partial x^\star(\theta)]^\top (\nabla_x L(x^\star(\theta), \theta)) + \nabla_\theta L(x^\star(\theta), \theta). \tag{14.30}$$

Using the implicit function theorem one can compute the Jacobian as

$$\partial x^\star(\theta) = -\left(\frac{\partial h}{\partial x}(x^\star(\theta), \theta)\right)^{-1} \frac{\partial h}{\partial \theta}(x^\star(\theta), \theta).$$

In practice, one replace in these formulas $x^\star(\theta)$ by x_t, which produces an approximation of $\nabla f(\theta)$. The disadvantage of this method is that it requires the resolution of a linear system, but its advantage is that it bypass the memory storage issue of the backpropagation algorithm.

Argmin layers One can define a mapping from some parameter θ to a point $x(\theta)$ by solving a parametric optimization problem

$$x(\theta) = \operatorname*{argmin}_x \mathcal{E}(x, \theta).$$

The simplest approach to solve this problem is to use a gradient descent scheme, $x_0 = 0$ and

$$x_{k+1} = x_k - \tau \nabla \mathcal{E}(x_k, \theta). \tag{14.31}$$

This has the form (14.25) when using the vector field $u(x, \theta) = \nabla \mathcal{E}(x_k, \theta)$.

Using formula (14.30) in this case where $h = \nabla \mathcal{E}$, one obtains

$$\nabla f^\star(\theta) = -\left(\frac{\partial^2 \mathcal{E}}{\partial x \partial \theta}(x^\star(\theta), \theta)\right)^\top \left(\frac{\partial^2 \mathcal{E}}{\partial x^2}(x^\star(\theta), \theta)\right)^{-1} (\nabla_x L(x^\star(\theta), \theta)) + \nabla_\theta L(x^\star(\theta), \theta)$$

In the special case where the function $f(\theta)$ is the minimized function itself, i.e. $f(\theta) = \mathcal{E}(x^\star(\theta), \theta)$, i.e. $L = \mathcal{E}$, then one can apply the implicit function theorem formula (14.30), which is much simpler since in this case $\nabla_x L(x^\star(\theta), \theta) = 0$ so that

$$\nabla f^\star(\theta) = \nabla_\theta L(x^\star(\theta), \theta). \tag{14.32}$$

This result is often called Danskin theorem or the envelope theorem.

Sinkhorn's algorithm Sinkhorn algorithm approximates the optimal distance between two histograms $a \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$ using the following recursion on multipliers, initialized as $x_0 \overset{\text{def.}}{=} (u_0, v_0) = (1_n, 1_m)$

$$u_{k+1} = \frac{a}{K v_k}, \quad \text{and} \quad v_{k+1} = \frac{b}{K^\top u_k}.$$

where $\div$ is the pointwise division and $K \in \mathbb{R}_+^{n \times m}$ is a kernel. Denoting $\theta = (a, b) \in \mathbb{R}^{n+m}$ and $x_k = (u_k, v_k) \in \mathbb{R}^{n+m}$, the OT distance is then approximately equal to

$$f(\theta) = \mathcal{E}(x_t, \theta) \overset{\text{def.}}{=} \langle a, \log(u_t) \rangle + \langle b, \log(v_t) \rangle - \varepsilon \langle K u_t, v_t \rangle.$$

Sinkhorn iteration are alternate minimization to find a minimizer of $\mathcal{E}$.

Denoting $\mathcal{K} \overset{\text{def.}}{=} \begin{pmatrix} 0 & K \\ K^\top & 0 \end{pmatrix} \in \mathbb{R}^{(n+m)\times(n+m)}$, one can re-write these iterations in the form (14.27) using

$$h(x, \theta) = \frac{\theta}{\mathcal{K} x} \quad \text{and} \quad L(x_t, \theta) = \mathcal{E}(x_t, \theta) = \langle \theta, \log(x_t) \rangle - \varepsilon \langle K u_t, v_t \rangle.$$

One has the following differential operator

$$[\partial_x h(x, \theta)]^\top = -\mathcal{K}^\top \operatorname{diag}\left(\frac{\theta}{(\mathcal{K} x)^2}\right), \quad [\partial_\theta h(x, \theta)]^\top = \operatorname{diag}\left(\frac{1}{\mathcal{K} x}\right).$$

Similarly as for the argmin layer, at convergence $x_k \to x^\star(\theta)$, one finds a minimizer of $\mathcal{E}$, so that $\nabla_x L(x^\star(\theta), \theta) = 0$ and thus the gradient of $f^\star(\theta) = \mathcal{E}(x^\star(\theta), \theta)$ can be computed using (14.32) i.e.

$$\nabla f^\star(\theta) = \log(x^\star(\theta)).$$

Bibliography

[1] E. Candès and D. Donoho. New tight frames of curvelets and optimal representations of objects with piecewise C^2 singularities. *Commun. on Pure and Appl. Math.*, 57(2):219–266, 2004.

[2] E. J. Candès, L. Demanet, D. L. Donoho, and L. Ying. Fast discrete curvelet transforms. *SIAM Multiscale Modeling and Simulation*, 5:861–899, 2005.

[3] S.S. Chen, D.L. Donoho, and M.A. Saunders. Atomic decomposition by basis pursuit. *SIAM Journal on Scientific Computing*, 20(1):33–61, 1999.

[4] P. L. Combettes and V. R. Wajs. Signal recovery by proximal forward-backward splitting. *SIAM Multiscale Modeling and Simulation*, 4(4), 2005.

[5] I. Daubechies, M. Defrise, and C. De Mol. An iterative thresholding algorithm for linear inverse problems with a sparsity constraint. *Commun. on Pure and Appl. Math.*, 57:1413–1541, 2004.

[6] D. Donoho and I. Johnstone. Ideal spatial adaptation via wavelet shrinkage. *Biometrika*, 81:425–455, Dec 1994.

[7] Heinz Werner Engl, Martin Hanke, and Andreas Neubauer. *Regularization of inverse problems*, volume 375. Springer Science & Business Media, 1996.

[8] M. Figueiredo and R. Nowak. An EM Algorithm for Wavelet-Based Image Restoration. *IEEE Trans. Image Proc.*, 12(8):906–916, 2003.

[9] Simon Foucart and Holger Rauhut. *A mathematical introduction to compressive sensing*, volume 1. Birkhäuser Basel, 2013.

[10] Stephane Mallat. *A wavelet tour of signal processing: the sparse way*. Academic press, 2008.

[11] D. Mumford and J. Shah. Optimal approximation by piecewise smooth functions and associated variational problems. *Commun. on Pure and Appl. Math.*, 42:577–685, 1989.

[12] Gabriel Peyré. *L'algèbre discrète de la transformée de Fourier*. Ellipses, 2004.

[13] J. Portilla, V. Strela, M.J. Wainwright, and Simoncelli E.P. Image denoising using scale mixtures of Gaussians in the wavelet domain. *IEEE Trans. Image Proc.*, 12(11):1338–1351, November 2003.

[14] L. I. Rudin, S. Osher, and E. Fatemi. Nonlinear total variation based noise removal algorithms. *Phys. D*, 60(1-4):259–268, 1992.

[15] Otmar Scherzer, Markus Grasmair, Harald Grossauer, Markus Haltmeier, Frank Lenzen, and L Sirovich. *Variational methods in imaging*. Springer, 2009.

[16] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[17] Jean-Luc Starck, Fionn Murtagh, and Jalal Fadili. *Sparse image and signal processing: Wavelets and related geometric multiscale analysis*. Cambridge university press, 2015.